Passive Mid-Stream Monitoring of Real-Time Properties

Lalita Jategaonkar Jagadeesan
Bell Laboratories Research, Lucent Technologies
2701 Lucent Lane
Lisle, IL 60532, USA
lalita@lucent.com

Ramesh Viswanathan
Bell Laboratories Research, Lucent Technologies
101 Crawfords Corner Rd.
Holmdel, NJ 07733, USA
rv@lucent.com

ABSTRACT

Passive monitoring or testing of complex systems and networks running in the field can provide valuable insights into their behavior in actual environments of use. In certain contexts, such as network management and intrusion detection for security, passive monitoring is the most applicable methodology for assuring correctness of the system's behavior. More generally, it can serve to complement and extend functional testing and fault detection efforts that take place during the software/product development lifecycle. Two distinguishing aspects of passive monitoring are that: (a) the fault detection process cannot influence the execution of the system by providing particular inputs to the system, and (b) observations are obtained mid-stream, from an unknown state in the middle of the execution of the system. In this paper, we present results on passively testing for *real-time* behavioral properties that can be applied to a large class of systems including those that can be modeled as timed automata. Our results provide a natural extension of the passive testing study conducted in [17] for untimed properties. We have implemented our approach using the real-time model checker UPPAAL, and we report on its application to passively test fault tolerance software in a telecommunications switch developed at Lucent Technologies.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.4 [**Software Engineering**]: Software/Program Verification; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Verification

Keywords

passive testing, run-time verification, monitoring, timed automata

EMSOFT'05, September 19–22, 2005, Jersey City, New Jersey, USA.

1. INTRODUCTION

As today's networks become increasingly complex and are comprised of equipment from multiple vendors, exhaustively testing all possible behaviors of the network – even under specified deployment environments – becomes infeasible. The analysis of *real-time* behavior additionally and significantly increases this complexity. Consequently, passive monitoring or testing of network equipment running in the field can provide valuable insights into their behavior in actual environments of use. For certain applications, such as network management [20] and intrusion detection for security [18], passive monitoring is, in fact, the only available methodology for validating the system's behavior. More generally, it can be used to complement and augment functional testing and fault detection efforts that take place during the product/software development lifecycle. Since passive monitoring observes network equipment while it is running in the field, a key challenge is that observations of the system are necessarily obtained *mid-stream*, from an unknown state during the middle of the execution of the system. Hence, passive monitoring techniques cannot make assumptions about the previous (unobserved) history of the system during its execution. Furthermore, they cannot influence the execution of the system by providing particular inputs to the system.

In this paper, we present an approach for passive monitoring of real-time properties, by observing the mid-stream behavior of systems during their execution in arbitrary environments. More precisely, we identify a class of system models that we call *expressively sufficient* with the implementation being tested assumed to be some instance of an expressively sufficient model. The conditions that we require of an expressively sufficient model are fairly simple and non-restrictive; popular formalisms such as timed automata [2] and timed transition systems [11] are expressively sufficient. For any correctness property $\mathcal{P}$ of an expressively sufficient model, we define a language (set) $PT(\mathcal{P})$ that consists of traces (including timing information) that can be exhibited by a correct implementation from some point mid-stream in its execution; passively testing for a property $\mathcal{P}$ can then be formulated as the problem of determining membership of the mid-stream observation in the language $PT(\mathcal{P})$. We present such passive testing algorithms for two general classes of properties: (a) conformance to a specification model, and (b) trace-containment in a specification language of correct traces. These algorithms are PSPACE in the size of the specification model or an automaton used to specify the language of correct traces.

In general, the mid-stream nature of passive testing places a limitation on the extent to which a property can be tested. For an arbitrary correctness property, while any rejected implementation is definitively faulty, all faulty implementations may not necessarily be detectable on the basis of their mid-stream observations. We therefore consider those properties for which complete fault coverage can be achieved through passive testing, and term such a property as being *passively testable*. For passively testable properties, an implementation is correct if and only if it is never rejected on the basis of its passively monitored mid-stream observations. We provide an exact characterization (necessary and sufficient conditions) of passively testable properties. Serendipitiously, this characterization enables the development of more efficient testing algorithms for passively testable properties. For example, when a specification machine is such that conformance to it is a passively testable property, the resulting passive testing algorithm is in NP when the specification machine is a timed automaton and polynomial time when the specification machine is an event-clock automaton [3]. Our characterization of passively testable properties therefore provides a systematic basis for identifying a subclass that can be passively tested completely and efficiently.

We have implemented our passive testing algorithms using the real-time model checker UPPAAL [7]. As a case study, we consider the application of our approach to passively monitor a fault tolerance software that is an important component of a telecommunications switch developed at Lucent Technologies. Our study reveals that most correctness properties of interest in this setting meet our characterization of being passively testable and can therefore be efficiently and completely tested through passive mid-stream monitoring.

An important technical aspect of our work is the manner in which timing information is represented in observed traces. Most previous work consider *timed traces* which are sequences of the form $\langle a_1, t_1\rangle \ldots \langle a_n, t_n\rangle$ denoting the sequential occurrence of events $a_1, \ldots, a_n$ tagged with their respective occurrence times $t_1, \ldots, t_n$. Here, we introduce an alternative notion of trace that we call *interval-timed traces* which are sequences of the form $t_1 a_1 t_2 a_2 \ldots t_n a_n t_{n+1}$ denoting that after an elapse of time t_1, the first event a_1 occurs, followed by an elapse of time interval t_2 after which a_2 occurs and so forth until the last event a_n after which a time interval t_{n+1} elapses during which no other event occurs. Besides the fact that timing information is represented in the form of time-intervals between successive occurrences of events, the other significant aspect in which interval-timed traces differ from timed traces is the inclusion of the last interval t_{n+1} which is not associated with the occurrence of any event. This permits a more direct and natural representation of the intuitions in the setting of passive testing. In particular, when monitoring a timed implementation from mid-stream in its execution, one cannot infer the absolute time occurrences of any observed events but one can observe the time intervals that are specified in an interval-timed trace. Furthermore, the time-interval observed to have elapsed after the occurrence of the last event during which no further events are observed is important to detecting violations of certain properties such as bounded liveness, *e.g.*, that a response must occur within 5 seconds of a request.

Our adoption of interval-timed traces facilitates a technically appealing development of our results in that they form a natural extension of those for the untimed setting described in [17]. Over interval-timed traces, we can define the notions of *timed concatenation*, *timed prefix*, and *timed suffix*; the passive testing results in this paper then generalize those in the untimed setting [17] by replacing the roles of concatenation, prefix and suffix (applicable to untimed traces) by their timed counterparts (over interval-timed traces).

1.1 Related work

While we have implemented our passive monitoring algorithms using the real-time model checker UPPAAL, our approach differs significantly from real-time model checking techniques [1]. In particular, real-time model checking assumes that a specification of the system is given as a (concurrent set of) timed automata; model checking tools then exhaustively analyze this specification against desired properties expressed in real-time variants of temporal logic. While model checking has been successful in identifying errors in critical portions of systems, it suffers from two challenges: it does not scale to large-scale systems because the size of the state space becomes intractable, and it identifies errors in the specification of the system rather than in the actual system implementation. These challenges are further exacerbated for analysis of real-time systems. In contrast, our passive monitoring approach analyzes executions of the actual implementation of real-time systems. However, it is not intended to perform exhaustive analysis.

Our approach also differs from conformance testing, and is intended to complement and augment conformance testing activities that take place during the product/software development cycle. In conformance testing, the system implementation under test is given as a black box, in which the interface to the black box is known, inputs can be sent to the black box, and the outputs of the black box are observable. A modelling formalism is chosen such that the implementation can be assumed to be an (unknown) model in this formalism, and the specification of the system can be described as a model in this formalism. Conformance testing then generates a comprehensive suite of test cases to validate that the implementation satisfies the specification; in particular, the test cases correspond to sequences of inputs sent to the implementation, and the resulting outputs are compared to those expected by the specification as per a formally specified conformance relation. The test case generation can be done prior to testing (off-line), or during testing (online/on-the-fly). Conformance testing for timed systems has been described in [14, 8, 16, 9], in which the modelling formalism corresponds to timed automata. Implicit in all of these approaches is that each test case executes the system from its initial state. Our approach differs from this work in two ways. First, it does not assume that all observations start from the initial state of the system, but rather that observations can be mid-stream. Second, it does not assume that the testing infrastructure can control the inputs to the system, but rather that the operational environment controls all inputs to the system: in this aspect, it is passively monitoring the system as opposed to actively testing it.

Our approach is closely related to run-time verification [6, 13, 19], in that it observes the behavior of systems during run-time, and analyzes the observed behavior against specifications of the intended real-time behavior of the system.

However, prior work in run-time verification of timed systems [6, 13, 19] implicitly assumes that all observed traces are generated from the initial state of the system, and hence that the system is typically reset to its initial state between observations. In practice, this is infeasible for large networks or systems, in which re-initializing the system can take a prohibitive amount of time. Our work, in contrast, supports the analysis of mid-stream observations, taken while the system is in the middle of its execution. Interestingly, our characterization of passively testable real-time properties shows it would be sound to check them directly against passively monitored traces as if they were observed from the initial state, oblivious of their mid-stream nature. A surprising consequence of our work is therefore the identification of a class of real-time properties for which the run-time verification tools of [6, 13, 8] and the timed observer infrastructure of [8, 16], all suitably extended to interval-timed traces, can be used for passive mid-stream monitoring as well.

The remainder of the paper is organized as follows. Section 2 presents the formal underpinnings of our approach. An overview of the fault tolerance software in a Lucent telecommunications switch on which we applied our approach is presented in Section 3. In Section 4, we describe the application of our approach to this fault tolerance software. Section 5 presents conclusions and future work.

2. FORMAL FOUNDATIONS

2.1 Preliminaries: Timed Automata

Timed Automata, introduced in [2], provide a simple yet powerful mechanism for formally specifying the behavior of systems that are governed by timing constraints, and is probably the most widely accepted (*cf.*[5]) and studied abstraction of real-time systems with finite control. Our presentation of passive testing for real-time systems builds upon the existing theory of timed automata in two ways. First, we use timed automata as a concrete example of a highly expressive formalism for modeling the system implementations (being passively tested) to which our results are applicable. Second, our passive testing algorithms are based on existing solutions to decision problems on languages of traces (that include timing information) expressible by timed automata. These two uses require slight differences in the details of the definition of timed automata. In specifying languages of traces, it is useful to include acceptance conditions that identify a subset of the generated traces as belonging to the language; these acceptance conditions do not however have any natural role when modeling system implementations. To distinguish between the two, we use the term *timed state machines* to refer to the formalism used for modeling system implementations and *timed automata* for expressing trace languages. We present a trace-semantics of timed transition systems consisting of *interval-timed traces* as described in Section 1, as opposed to classical timed traces. The move to interval-timed traces also allows us to introduce a slight enrichment in the definition of timed automata without increasing the algorithmic complexity of the decision problems of interest.

Timed State Machines are finite state machines augmented with a finite set of real-valued clock variables. The states in a timed state machine are called locations. Time can elapse in a location while transitions are instantaneous. Any of the clock variables can be reset to zero simultaneously with any transition. At any instant, the reading of a clock variable equals the time elapsed since the last time it was reset. Each transition includes a clock constraint that must be satisfied by the current values of the clock variables for the transition to be taken. Each location is associated with a clock constraint called its invariant; time can elapse in a location only as long as the values of the clock variables continue to satisfy its invariant. The clock constraints used in transitions and as invariants are as follows. Let X be a finite set of clocks. The set of clock constraints over X, denoted $\Phi(X)$, includes formulas defined by the grammar $\phi ::= \ x \sim c \mid \phi_1 \wedge \phi_2$ where $x \in X$, c is a constant in $\mathbf{Q}$, the set of non-negative rationals, and $\sim$ is in $\{\leq, \geq, <, >\}$. The precise definition of timed state machines now follows.

Definition 2.1 (Timed State Machines). *A* timed state machine $\mathcal{M}$ *is a tuple* $\langle L, L^o, \Sigma, X, \gamma, E\rangle$*, where* L *is a finite set of locations,* $L^o \subseteq L$ *is the set of initial locations,* Σ *is a finite set of labels,* X *is a finite set of clocks,* $\gamma: L \rightarrow \Phi(X)$ *is a mapping with* $\gamma(l)$ *giving the invariant associated with location* l*, and* $E \subseteq L \times \Sigma \times 2^X \times \Phi(X) \times L$ *is a set of edges. An edge* $\langle s, a, \lambda, \phi, s'\rangle \in E$ *represents an* a*-labeled transition from location* s *to location* s'*,* ϕ *is a clock constraint that specifies when the transition is enabled, and* λ *is the set of clocks that are reset to zero on this transition.*

A clock valuation ν over a set of clocks X is a mapping from X to $\mathbf{Q}$. For $t \in \mathbf{Q}$, the clock valuation $\nu + t$ is defined by $(\nu + t)(x) = \nu(x) + t$ for every $x \in X$. For $\lambda \subseteq X$, the clock valuation $\nu[\lambda := 0]$ is defined as mapping any $x \in \lambda$ to 0 and any $x \notin \lambda$ to $\nu(x)$. Definition 2.2 below defines the semantics of a timed state machine with alphabet Σ as a subset of $(\mathbf{Q}\Sigma)^*\mathbf{Q}$, *i.e.*, a set of sequences of the form $t_1 a_1 \ldots t_n a_n t_{n+1}$, where $n \geq 0$, $a_i \in \Sigma$ for $1 \leq i \leq n$ and $t_i \in \mathbf{Q}$ for $1 \leq i \leq n+1$.[1]

Definition 2.2 (Interval-Timed Trace Semantics). *A* run *of a timed state machine* $\mathcal{M} = \langle L, L^o, \Sigma, X, \gamma, E\rangle$ *is an edge-labelled finite sequence of the form* $\langle s_0, \nu_0\rangle \xrightarrow{t_1} \langle s_0, \nu_0'\rangle \xrightarrow{a_1} \langle s_1, \nu_1\rangle \cdots \langle s_i, \nu_i\rangle \xrightarrow{t_{i+1}} \langle s_i, \nu_i'\rangle \xrightarrow{a_{i+1}} \langle s_{i+1}, \nu_{i+1}\rangle \cdots \langle s_n, \nu_n\rangle \xrightarrow{t_{n+1}} \langle s_n, \nu_n'\rangle$*, where* $n \geq 0$ *and* $s_i \in L$*,* ν_i, ν_i' *are clock valuations over* X *for* $0 \leq i \leq n$*,* $a_i \in \Sigma$ *for* $1 \leq i \leq n$*, and* $t_i \in \mathbf{Q}$ *for* $1 \leq i \leq n+1$ *such that*

- $s_0 \in L^o$ *and* $\nu_0(x) = 0$ *for all* $x \in X$
- *For all* $t' \in \mathbf{Q}$ *such that* $0 \leq t' \leq t_{i+1}$*,* $\nu_i + t'$ *satisfies* $\gamma(s_i)$ *and* $\nu_i' = \nu_i + t_{i+1}$*, for* $0 \leq i \leq n$
- *There is some transition* $\langle s_i, a_{i+1}, \lambda, \phi, s_{i+1}\rangle \in E$ *such that* ν_i' *satisfies* ϕ *and* $\nu_{i+1} = \nu_i'[\lambda := 0]$*, for* $0 \leq i < n$

The trace *of a run* $\langle s_0, \nu_0\rangle \xrightarrow{t_1} \langle s_0, \nu_0'\rangle \xrightarrow{a_1} \langle s_1, \nu_1\rangle \cdots \langle s_i, \nu_i\rangle \xrightarrow{t_{i+1}} \langle s_i, \nu_i'\rangle \xrightarrow{a_{i+1}} \langle s_{i+1}, \nu_{i+1}\rangle \cdots \langle s_n, \nu_n\rangle \xrightarrow{t_{n+1}} \langle s_n, \nu_n'\rangle$ *is the sequence of its edge labels* $t_1 a_1 \ldots t_n a_n t_{n+1}$*. The* trace semantics *of the timed state machine,* $[\![M]\!]$ *is defined to be the set of traces of all runs of the machine* M*.*

A timed automaton is a timed state machine augmented with accepting locations and acceptance invariants associated with each accepting location. An accepting run is one

[1]We use rationals rather than reals because they correspond better to practically measurable intervals; however, the theoretical development is independent of this choice.

in which the last location is an accepting location and any time elapsed in the last location meets its acceptance invariant. The set of traces of accepting runs is the language (of interval-timed traces) accepted by a timed automaton.

DEFINITION 2.3 (TIMED AUTOMATA). *A* timed automaton $\mathcal{A}$ *is a tuple* $\langle L, L^o, \Sigma, X, \gamma, E, L^F, \gamma^F \rangle$, *where* L, L^o, Σ, X, γ, E *are as in Definition 2.1,* $L^F \subseteq L$ *is a set of accepting locations, and* $\gamma^F : L^F \rightarrow \Phi(X)$ *gives the accepting invariant associated with each accepting location. An* accepting run *of* $\mathcal{A}$ *is a run* $\langle s_0, \nu_0 \rangle \stackrel{t_1}{\rightarrow} \langle s_0, \nu_0' \rangle \stackrel{a_1}{\rightarrow} \langle s_1, \nu_1 \rangle \cdots \langle s_i, \nu_i \rangle \stackrel{t_{i+1}}{\rightarrow} \langle s_i, \nu_i' \rangle \stackrel{a_{i+1}}{\rightarrow} \langle s_{i+1}, \nu_{i+1} \rangle \cdots \langle s_n, \nu_n \rangle \stackrel{t_{n+1}}{\rightarrow} \langle s_n, \nu_n' \rangle$ *of the timed state machine* $\langle L, L^o, \Sigma, X, \gamma, E \rangle$ *such that* $s_n \in L^F$ *and* ν_n' *satisfies* $\gamma^F(s_n)$. *The language accepted by the timed automaton,* $L(\mathcal{A})$ *is the set of traces of all accepting runs of* $\mathcal{A}$.

Timed automata, as previously studied, do not include the accepting invariant γ^F and their languages are defined to be sets of timed traces that are of the form $\langle a_1, t_1 \rangle, \ldots, \langle a_n, t_n \rangle$ where $n \geq 0$; for clarity of distinction, we use $L^T(\mathcal{A})$ to denote the language of timed traces of a timed automaton without the invariant γ^F. The decision problems of interest to us that have been previously studied are the membership problem of whether a timed trace σ belongs to the timed language $L^T(\mathcal{A})$ of a timed automaton which is in NP [4], and the emptiness problem of whether the timed language $L^T(\mathcal{A})$ of a timed automaton is empty which is in PSPACE [2, 5]. We now show that the corresponding decision problems for the interval-timed language given in Definition 2.3 fall into the same complexity classes. For any alphabet Σ, define the alphabet $\Sigma^T = \Sigma \cup \{\surd\}$ where $\surd \notin \Sigma$; intuitively, the extra label $\surd$ is used to represent the end of a trace. For any interval-timed trace $\sigma = t_1 a_1 \ldots t_n a_n t_{n+1}$ over the alphabet Σ, define the timed trace σ^T over the alphabet Σ^T as the sequence $\langle a_1, t_1 \rangle, \ldots, \langle a_i, t_1 + \cdots + t_i \rangle, \langle a_n, t_1 + \cdots + t_n \rangle, \langle \surd, t_1 + \cdots + t_{n+1} \rangle$. For any timed automaton $\mathcal{A}$ of Definition 2.3, we can obtain a corresponding timed automaton without acceptance invariants $\mathcal{A}^T$ by adding a new location that is the single accepting location and adding transitions from all accepting locations l of $\mathcal{A}$ on the special symbol $\surd$ that are required to meet the clock invariant $\gamma^F(l)$. That is, for $\mathcal{A} = \langle L, L^o, \Sigma, X, \gamma, E, L^F, \gamma^F \rangle$, we define the timed automaton (without acceptance invariants) $\mathcal{A}^T$ as $\langle L \cup \{f\}, L^o, \Sigma^T, X, \gamma, E', \{f\} \rangle$ where f is a location not in L, and $E' = E \cup \{\langle l, \surd, \emptyset, \gamma^F(l), f \rangle \mid l \in L^F\}$. It can be easily verified that for any interval-timed trace σ over the alphabet Σ, we have that $\sigma \in L(\mathcal{A})$ iff $\sigma^T \in L^T(\mathcal{A}^T)$. Since the construction of σ^T and $\mathcal{A}^T$ are linear in σ and $\mathcal{A}$ respectively, it follows that the membership problem for interval-timed traces is also in NP. Additionally, one can verify that any timed trace $\sigma' \in L^T(\mathcal{A}^T)$ must be of the form σ^T for some interval-timed trace σ. It therefore follows that $L(\mathcal{A}) = \emptyset$ iff $L^T(\mathcal{A}^T) = \emptyset$ which yields a PSPACE algorithm for the emptiness problem for interval-timed traces.

A timed state machine can be viewed as a timed automaton in which all locations are accepting locations and accepting invariants are the same as location invariants. That is, for a timed state machine $M = \langle L, L^o, \Sigma, X, \gamma, E \rangle$, we define its corresponding timed automaton $\mathcal{A}^M$ to be $\langle L, L^o, \Sigma, X, \gamma, E, L, \gamma \rangle$ which has the property that $L(\mathcal{A}^M) = [\![M]\!]$. We use $\mathcal{A}_1 \| \mathcal{A}_2$ to denote the standard product construction on timed automata $\mathcal{A}_1$ and $\mathcal{A}_2$ (with acceptance invariants on the pairs of final locations being the conjunction of the acceptance invariant of each component location), noting that with respect to their interval-timed trace languages given by Definition 2.3, we have that $L(\mathcal{A}_1 \| \mathcal{A}_2) = L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$.

2.2 A General Framework for Passive Testing

In this section, we formally define the passive testing problem for real-time systems. While timed state machines serve as our example model of the system implementations under test, our results are largely independent of the details of their definition. Therefore, rather than developing our results specifically for timed state machines, we develop a general axiomatization of implementation models for which our results hold. This axiomatization can be seen as a distillation of the key characteristics of timed state machines that enable our results and allows them to be applicable any other choice of real-time system model as long as it satisfies this axiomatization.

Recall that an interval-timed trace over a set Σ is an element of $(\mathbf{Q}\Sigma)^*\mathbf{Q}$, *i.e.*, a sequence of the form $t_1 a_1 \ldots t_n a_n t_{n+1}$ where $n \geq 0$, $a_i \in \Sigma$ for $1 \leq i \leq n$, and $t_i \in \mathbf{Q}$ for $1 \leq i \leq n+1$. Interval-timed traces permit a natural definition of a *timed concatenation* operation that corresponds to the consecutive occurrence of two traces. For interval-timed traces $\sigma = t_1 a_1 \ldots t_n a_n t_{n+1}$ and $\sigma' = t_1' a_1' \ldots t_m' a_m' t_{m+1}'$, the timed concatenation $\sigma \oplus \sigma'$ is defined to be the interval-timed trace $t_1 a_1 \ldots t_n a_n (t_{n+1} + t_1') a_1' \ldots t_m' a_m' t_{m+1}'$. The trace 0 acts as the identity of timed concatenation with $\sigma \oplus 0 = 0 \oplus \sigma = \sigma$ for any trace σ. We define the notion of *timed prefix* with respect to timed-concatenation, *i.e.*, we say that a trace σ is a timed prefix of a trace σ_1 if there exists a trace σ' such that $\sigma \oplus \sigma' = \sigma_1$. Note that timed prefixes are not the same as sequence prefixes, *e.g.*, the traces 0.5 and $1a1$ are timed prefixes of the trace $1a2.5b3$ neither of which are sequence prefixes.

Our assumptions on the formal model representing system implementations being tested are captured in the following definition of *expressively sufficient* models.

DEFINITION 2.4 (IMPLEMENTATION MODEL). *A formalism for modeling implementations is* expressively sufficient *if we can define functions* $\mathcal{O}(\cdot)$ *and* $[\![\cdot]\!]$ *on the class of all formal models with* $\mathcal{O}(I)$ *a set, and* $[\![I]\!] \subseteq (\mathbf{Q}O(I))^*\mathbf{Q}$, *for any model* I. *Further, the following two axioms must be satisfied:*

(Axiom I) *For any* $\sigma, \sigma' \in (\mathbf{Q}\mathcal{O}(I))^*\mathbf{Q}$, *if* $\sigma \oplus \sigma' \in [\![I]\!]$ *then* $\sigma \in [\![I]\!]$

(Axiom II) *For any set* Σ *and trace* $\sigma \in (\mathbf{Q}\Sigma)^*\mathbf{Q}$ *there is a model* I_σ *with* $\mathcal{O}(I_\sigma) = \Sigma$ *such that* $[\![I_\sigma]\!] = \{\ \sigma' \in (\mathbf{Q}\Sigma)^*\mathbf{Q} \mid \sigma' \oplus \sigma'' = \sigma$ *for some* $\sigma'' \in (\mathbf{Q}\Sigma)^*\mathbf{Q}\ \}$.

Intuitively, for an implementation model I, the set $\mathcal{O}(I)$ corresponds to the atomic observations or events that I can exhibit at any particular instant, such as a message being sent, or an input operation, or an output operation performed. These atomic observations are extended over time with timing information to yield interval-timed traces with $[\![I]\!]$ giving the set of all traces that can be observed from I upto any time instant. (Axiom I) then captures the intuitive requirement that if an implementation can exhibit the observed sequence $\sigma \oplus \sigma'$, then it can also exhibit the

timed prefix σ (namely, earlier in the same execution trace). (Axiom II) requires that the class of formal models be expressive enough so that for any interval-timed trace σ, there is an implementation I_σ that exhibits exactly σ and no other observations subsequently. (Axiom II) serves as an important technical requirement in the development of the results of Section 2.4.

As our canonical example, timed state machines satisfy the requirement of being *expressively sufficient*.

EXAMPLE 2.5. *For a timed state machine M given by $\langle L, L^o, \Sigma, X, \gamma, E\rangle$, we take $\mathcal{O}(M)$ to be Σ and $[\![M]\!]$ as given by Definition 2.2 which can be seen to satisfy (Axiom I). For (Axiom II), consider any sequence $\sigma = t_1 a_1 \ldots t_n a_n t_{n+1}$, where $n \geq 0$, $a_i \in \Sigma$ for $1 \leq i \leq n$ for some set Σ, and $t_i \in \mathbf{Q}$ for $1 \leq i \leq n+1$. We define the implementation $I_\sigma = \langle L, L^o, \Sigma, X, \gamma, E\rangle$, where $L = \{s_i \mid 0 \leq i \leq n\}$, $L^o = \{s_0\}$, $X = \{x\}$, $\gamma(s_i)$ is the constraint $x \leq t_{i+1}$ for $0 \leq i \leq n$, and $E = \{\langle s_i, a_{i+1}, \{x\}, x = t_{i+1}, s_{i+1}\rangle \mid 0 \leq i < n\}$. It is then easy to establish by induction that $[\![I_\sigma]\!]$ satisfies the requirement of (Axiom II). Note that the machine I_σ is deterministic and therefore the smaller class of deterministic timed state machines are expressively sufficient, as well.*

In the rest of this section, unless otherwise mentioned, we consider models with respect to any ambient modeling formalism that is expressively sufficient, and refer to models in this ambient formalism as implementations.

For an (unknown) implementation under test, we are interested in determining through passively monitoring its behavior, whether it satisfies some desired correctness property. For the purposes of our framework, it suffices to abstractly view any correctness property as a collection of implementations, namely those that satisfy the property.

DEFINITION 2.6. *A* property, *$\mathcal{P}$, is a collection of implementations. An implementation I satisfies a property $\mathcal{P}$, denoted $I \models \mathcal{P}$, if $I \in \mathcal{P}$.*

We now consider the question of how to passively test a black-box implementation for some property $\mathcal{P}$. In passive testing, we do not have access to the implementation itself, nor can we assume any knowledge of how long the implementation has already been running when we begin observing its execution. Instead, we monitor its execution starting from some time instant after which events/actions in the execution and their occurrence instants can be observed. The information available to the monitor can therefore be naturally represented as a sequence of the form $t_1 a_1 \ldots t_n a_n t_{n+1}$ denoting an observation over a time-interval of length $t_1 + \cdots + t_{n+1}$, with t_1 marking the time-interval from the instant monitoring is begun to the occurrence of the first observed event a_1, t_i's marking off the time-intervals between observation of the event a_{i-1} and the event a_i for $2 \leq i \leq n$, and t_{n+1} marking the time-interval after the last event a_n occurred. We assume that the monitor can measure time intervals only upto finite granularity or precision but not to arbitrary precision; to avoid assumptions on the precision of the digital clocks used by the monitor, which could be arbitrarily finite, we take the $t_1, \ldots, t_{n+1}$ to be rationals. We can therefore consider the passively monitored observation to be an interval-timed trace σ. We now define when the monitored implementation cannot be deemed faulty (*i.e.*, violating the property $\mathcal{P}$) on the basis of such an observed trace, σ. Since the observation was begun mid-stream, we have to assume that some unknown trace σ' preceded the observed sequence σ. Thus, the implementation cannot be deemed faulty if $\sigma' \oplus \sigma$ is a possible trace of a correct implementation, *i.e.*, an implementation I satisfying $\mathcal{P}$. We consider such traces σ to be acceptable when passively testing the property $\mathcal{P}$, and formally define $PT(\mathcal{P})$ to be the set of all such acceptable traces.

DEFINITION 2.7. *For a property $\mathcal{P}$, define the set $PT(\mathcal{P})$ of interval-timed traces* accepted by passive testing *as $\sigma \in PT(\mathcal{P})$ if and only if there is some implementation $I \models \mathcal{P}$ and trace σ' with $\sigma' \oplus \sigma \in [\![I]\!]$.*

Conversely, if we observe an interval-timed trace $\sigma \notin PT(\mathcal{P})$ at any point midstream, then we can conclude that the implementation under traces is definitely faulty; traces not in $PT(\mathcal{P})$ can therefore be considered fault-symptomatic.

PROPOSITION 2.8 (FAULT SYMPTOMATIC TRACES). *Let $\mathcal{P}$ be a property. For any implementation I, if we have $\sigma' \oplus \sigma \in [\![I]\!]$ with $\sigma \notin PT(\mathcal{P})$ then $I \not\models \mathcal{P}$.*

We therefore define a *passive testing algorithm* for some property $\mathcal{P}$ as determining membership of a given trace in $PT(\mathcal{P})$.

2.3 Passive Testing Algorithms for General Timed Properties

In this section, we present a passive testing algorithm for properties expressed in the form of requiring conformance to a specified timed state machine. In the untimed setting, conformance to a finite state machine was the first passive testing problem studied [15]. The proposed passive testing algorithm was called the homing algorithm because it began by assuming that the implementation could be in any state and then progressively refined this knowledge by trying to infer or "home" in to the current state of the specification machine on the basis of the events observed. In the timed setting, a similar presentation is more intricate because the state space consisting of locations and clock valuations is infinite. We therefore instead present the algorithm by a reduction to the emptiness problem for a suitably constructed timed automaton — the algorithm for checking emptiness then does the necessary "homing".

Let $\prec$ be a conformance relation, with $I \prec M$ denoting that an implementation I is conformant to the timed state machine M. The property of implementations that are conformant (w.r.t.$\prec$) to a (fixed) specification timed state machine M, denoted $\mathcal{P}_\prec^M$ is then the set $\{\ I \mid I \prec M\}$. The following proposition characterizes the set of accepted passively monitored traces for the property $\mathcal{P}_\prec^M$ for any conformance relation $\prec$ that admits the specification M itself as a conformant implementation and that is no weaker than trace containment. Almost all typical conformance relations, such as being timed simulable, timed bisimilarity, trace containment, and trace equivalence clearly meet these requirements.

PROPOSITION 2.9. *If a conformance relation $\prec$ is such that $M \prec M$ and for any implementation I, we have that $I \prec M \supset [\![I]\!] \subseteq [\![M]\!]$, then*

$$PT(\mathcal{P}_\prec^M) = \{\sigma \mid \exists \sigma'.\ \sigma' \oplus \sigma \in [\![M]\!]\}$$

For an interval-timed trace σ over an alphabet Σ, we now construct a timed automaton $\mathcal{A}^\sigma$ that accepts exactly

traces of the form $\sigma' \oplus \sigma$ for any $\sigma' \in (\mathbf{Q}\Sigma)^*\mathbf{Q}$. For $\sigma = t_1 a_1 \ldots t_n a_n t_{n+1}$ with $n \geq 0$, the automaton $\mathcal{A}^\sigma$ is defined to be $\langle L, L^o, \Sigma, X, \gamma, E, L^F, \gamma^F \rangle$, where $L = \{s_i \mid 0 \leq i \leq n\}$, $L^o = \{s_0\}$, $X = \{x\}$, $\gamma(s_i)$ is the vacuous constraint $x \geq 0$ for all $0 \leq i \leq n$, $E = \{\langle s_0, a, \{x\}, x \geq 0, s_0\rangle \mid a \in \Sigma\} \cup \{\langle s_0, a_1, \{x\}, x \geq t_1, s_1\rangle\} \cup \{\langle s_i, a_{i+1}, \{x\}, x = t_{i+1}, s_{i+1}\rangle \mid 1 \leq i < n\}$, $L^F = \{s_n\}$, and $\gamma^F(s_n)$ is the constraint that $x = t_{n+1}$. The automaton $\mathcal{A}^\sigma$ contains a single clock x that is reset to 0 with every transition. The transitions in $\mathcal{A}^\sigma$ are of three kinds. The self-loops on the initial state s_0 for every $a \in \Sigma$ together with the clock constraint $(x \geq t_1)$ on the a_1-labelled transition from s_0 to s_1 ensure that all possible prefixes are time-concatenated with $t_1 a_1$ to reach s_1, and the transitions from s_1 onwards ensure that the only accepting trace from s_1 is $t_2 a_2 \ldots t_n a_n t_{n+1}$. It is worth noting the use of the accepting invariant γ^F in ensuring that the last time-interval elapsed is exactly t_{n+1}, which motivates our introduction of accepting invariants in timed-automata (Definition 2.3). We can therefore establish that the automaton $\mathcal{A}^\sigma$ accepts all timed prefix extensions of σ.

PROPOSITION 2.10. *For any trace $\sigma \in (\mathbf{Q}\Sigma)^*\mathbf{Q}$, the timed automaton $\mathcal{A}^\sigma$ has the property that*

$$L(\mathcal{A}^\sigma) = \{\sigma' \oplus \sigma \mid \sigma' \in (\mathbf{Q}\Sigma)^*\mathbf{Q}\}$$

Recalling from Section 2.1 the properties of the timed automaton $\mathcal{A}^M$ (for a timed state machine M) and product construction, it therefore follows from Proposition 2.10 that $L(\mathcal{A}^\sigma \| \mathcal{A}^M) = \{\sigma' \oplus \sigma \in [\![M]\!] \mid \sigma' \in (\mathbf{Q}\Sigma)^*\mathbf{Q}\}$. This combined with Proposition 2.9 therefore yields the following corollary.

COROLLARY 2.11. *For any timed state machine M and any conformance relation $\prec$ satisfying the conditions of Proposition 2.9, we have that*

$$\sigma \in PT(\mathcal{P}^M_\prec) \quad \textit{iff } L(\mathcal{A}^\sigma \| \mathcal{A}^M) \neq \emptyset$$

Corollary 2.11 gives a passive testing algorithm for the conformance problem based on an algorithm for the language emptiness problem for timed automata. The size of $\mathcal{A}^\sigma$ is linear in the size of Σ and the length of σ. The size of the product automaton is $O(nm)$ where n is the maximum of length of σ and cardinality of Σ and m is the size of M. The emptiness problem for timed automata expressing interval-timed traces was established to be in PSPACE in Section 2.1 and therefore this passive testing algorithm is PSPACE (in the size of σ and M).

2.4 Passively Testable Timed Properties

In passively testing an implementation for some property $\mathcal{P}$, we deem the correctness of the implementation (*i.e.*, whether it satisfies $\mathcal{P}$) on the basis of membership of its passively observed traces in $PT(\mathcal{P})$. By Proposition 2.8, if the implementation is rejected, *i.e.*, it exhibits a passively observed trace that is not in $PT(\mathcal{P})$, then the implementation is definitely faulty. However, as the following example shows, the converse does not hold in general, *i.e.*, an accepted implementation is not necessarily correct.

EXAMPLE 2.12. *Let Σ be some alphabet, δ some constant in $\mathbf{Q}$, and define the language L of interval-timed traces over Σ as $L = \{t_1 a_1 \ldots a_n t_{n+1} \mid t_1 \leq \delta\}$, and consider the property $\mathcal{P} = \{I \mid [\![I]\!] \subseteq L\}$. Intuitively, $\mathcal{P}$ is the property of implementations whose first action occurs within δ time units. We show that any interval-timed trace $\sigma \in PT(\mathcal{P})$. Let σ' be the trace $0a0$ for some $a \in \Sigma$. Then for the implementation $I = I_{\sigma' \oplus \sigma}$ given by Axiom (II) (of Definition 2.4), we have that $I \models \mathcal{P}$ and $\sigma' \oplus \sigma \in [\![I]\!]$. Therefore, by Definition 2.7, $\sigma \in PT(\mathcal{P})$. Let τ be some trace of the form $t_1 a_1 \ldots a_n t_{n+1}$ where $t_1 > \delta$ and consider the implementation I_τ given by Axiom (II) of Definition 2.4. Clearly, $\tau \notin L$ and hence $I_\tau \not\models \mathcal{P}$. However, because $PT(\mathcal{P}) = (\mathbf{Q}\Sigma)^*\mathbf{Q}$, we trivially have that for any σ', σ with $\sigma' \oplus \sigma \in [\![I_\tau]\!]$ $\sigma \in PT(\mathcal{P})$. Thus, the implementation I_τ would never be rejected when being passively tested for property $\mathcal{P}$ even though $I_\tau \not\models \mathcal{P}$.*

In this section, we therefore explore the class of properties for which, in addition to guaranteeing the faultiness of rejected implementations, passive testing can be used to establish the correctness of accepted ones as well, *i.e.*, if every passively monitored trace of an implementation is accepted then the implementation is assured to satisfy the property. We define such properties as being *passively testable*.

DEFINITION 2.13 (PASSIVE TESTABILITY). *A property $\mathcal{P}$ is* passively testable *if and only if it has the following closure property: Suppose that I is an implementation such that for all traces σ', σ with $\sigma' \oplus \sigma \in [\![I]\!]$, we have that $\sigma \in PT(\mathcal{P})$. Then $I \models \mathcal{P}$.*

Definition 2.13 can be read, contrapositively, as stating that a passively testable property is one for which any faulty implementation $I \not\models \mathcal{P}$ manifests a fault-symptomatic trace and would therefore be rejected through passive testing. Passively testable properties therefore admit a complete fault coverage through passive testing.

We are interested in characterizing the class of properties that are passively testable. For a property $\mathcal{P}$, define the language $L_\mathcal{P}$ of interval-timed traces as $L_\mathcal{P} = \bigcup_{I \models \mathcal{P}} [\![I]\!]$. We begin by deriving an immediate consequence of Definition 2.13 (its proof appears in the appendix).

LEMMA 2.14. *Suppose that $\mathcal{P}$ is a passively testable property. Then $I \models \mathcal{P}$ iff $[\![I]\!] \subseteq L_\mathcal{P}$.*

Lemma 2.14 yields a necessary condition for a property to be passively testable, which can be restated in terms of the classical notion of safety. Define a language L of interval-timed traces to be *timed prefix-closed* if and only if for any σ, σ', if $\sigma' \oplus \sigma \in L$ then $\sigma' \in L$. We call a property $\mathcal{P}$ to be a *timed safety property* if there exists a timed prefix-closed language L such that $I \models \mathcal{P}$ iff $[\![I]\!] \subseteq L$. Using (Axiom I) of Definition 2.4, it is easy to see that the language $L_\mathcal{P}$ is timed prefix-closed for any property $\mathcal{P}$. We therefore obtain the following corollary of Lemma 2.14.

COROLLARY 2.15. *Suppose that $\mathcal{P}$ is a passively testable property. Then $\mathcal{P}$ is a timed safety property.*

Corollary 2.15 shows that the only passively testable properties are timed safety properties. However, not all timed safety properties are passively testable. The language L of Example 2.12 is timed prefix-closed and the property $\mathcal{P}$ considered in Example 2.12 is therefore a timed safety property that is not passively testable. Timed safety is therefore a necessary but not sufficient condition for passive testability. We next identify the set of necessary and sufficient conditions under which a property is passively testable. For a language L of interval-timed traces, we call L *timed suffix-closed* if and only if for any σ, σ', if $\sigma' \oplus \sigma \in L$ then $\sigma \in L$.

Theorem 2.16 below provides an exact characterization of passively testable properties and its proof (appearing in the appendix) makes pivotal use of the requirement of (Axiom II) of Definition 2.4.

THEOREM 2.16. *A property $\mathcal{P}$ is passively testable if and only if the following two conditions hold:*

1. *$I \models \mathcal{P}$ iff $[\![I]\!] \subseteq L_{\mathcal{P}}$*
2. *$L_{\mathcal{P}}$ is timed suffix-closed.*

Using Theorem 2.16, we can characterize passively testable properties as a natural special subclass of timed safety properties.

COROLLARY 2.17. *A property $\mathcal{P}$ is a passively testable property if and only if there is a timed prefix-closed and timed suffix-closed language L of interval-timed traces such that $I \models \mathcal{P}$ iff $[\![I]\!] \subseteq L$.*

While passively testable properties are a subclass of timed safety-properties, it should be noted that in the setting of interval-timed traces these include properties such as bounded liveness. For example, the property that every a should be followed by a b within a time-interval δ can be expressed as the language $L = \{t_1 a_1 \ldots a_n t_{n+1} \mid \forall i.a_i = a \Rightarrow (\forall j > i.t_{i+1} + \cdots + t_j > \delta \Rightarrow \exists i < k \leq j.a_j = b)\}$ which can be seen to be timed prefix- and timed suffix-closed.

The characterization of passively testable properties given by Theorem 2.16 leads to the following characterization of $PT(\mathcal{P})$ for passively testable properties $\mathcal{P}$ (proof in Appendix).

THEOREM 2.18. *Suppose that $\mathcal{P}$ is a passively testable property. Then $PT(\mathcal{P}) = L_{\mathcal{P}}$.*

The significant import of Theorem 2.18 is that it yields more efficient passive testing algorithms for passively testable properties. For example, applying Theorems 2.16 and 2.18 to trace-containment, we obtain the following corollary.

COROLLARY 2.19. *Consider any modeling formalism that is expressively sufficient and let M be a model in this formalism. Then the property $\mathcal{P}^M_{\preceq} = \{I \mid [\![I]\!] \subseteq [\![M]\!]\}$, of being trace-contained in M, is passively testable if and only if $[\![M]\!]$ is timed suffix-closed. If $[\![M]\!]$ is timed suffix-closed then $PT(\mathcal{P}^M_{\preceq}) = [\![M]\!]$.*

If the specification machine M is a timed state machine, then Corollary 2.19 yields a passive testing algorithm that consists of checking for membership in $L(\mathcal{A}^M)$ which can be done in NP, in contrast to the PSPACE algorithm presented in Section 2.3. If M is an event-clock automaton [3], the complexity reduces even more significantly to polynomial time. In general, the characterization of Theorem 2.18 allows checking the passively observed trace directly against the property specification behaving as if the trace was observed from the beginning without having to account for its mid-stream nature.

Finally, we present an alternative characterization of passive testability that is sometimes easier to establish for certain properties. This characterization is obtained by considering the complement of $L_{\mathcal{P}}$, the set of violating traces. A property is passively testable if and only if the set of violating traces satisfies the closure condition that any trace containing a violating subtrace must itself also be a violating trace.

COROLLARY 2.20. *A property $\mathcal{P}$ is passively testable iff the set $S = \overline{L_{\mathcal{P}}}$ satisfies the following closure property: If $\sigma \in S$ then for any σ', σ'' we have that $\sigma' \oplus \sigma \oplus \sigma'' \in S$.*

In particular, Corollary 2.20 implies that if a faulty subtrace is observed then no (unobserved) past behavior could provide mitigating circumstances to make the complete trace correct. This provides an intuitive explanation of the key characteristic of passively testable properties that makes them completely testable even in the absence of information about the initial unseen segment of the trace.

3. A TELECOMMUNICATIONS APPLICATION: THE HEART-BEAT MONITOR

As a case study, we have applied our approach to the "Heart-Beat Monitor" (HBM), a telephone switching application developed at Lucent Technologies. The HBM of a telephone switch determines the status of different elements connected to the switch by measuring propagation delays of messages transmitted via these elements. This information plays an important role in the routing of data in the switch, and can significantly impact switch performance.

In telephone switches, calls are typically routed through a network of hardware devices, involving a distributed set of processors. In order to ensure the reliability of calls, switches can determine the status of their processors by measuring propagation delays of messages transmitted between them. Longer than expected delays may indicate potential problems; the switch may then temporarily cease to connect new telephone calls over all hardware units connected to the offending processor. This can significantly impact switch performance if the switch normally relies on these units to carry a substantial proportion of telephone calls.

Using our approach, we have passively tested the "Heart-Beat Monitor" (HBM) of a Lucent switching system against its intended real-time properties. The HBM software is responsible for measuring the propagation delays of messages between two processors A and B. The HBM, running on processor A, periodically sends a "heart-beat" message to processor B. Upon receipt of the heart-beat message, processor B responds by sending an acknowledgment back to processor A. The HBM monitors the delay between the transmission and acknowledgment of messages. If such delays become unacceptable, HBM temporarily ceases the routing of all new telephone calls over processor B – this is termed as *resource suspension* in this paper.

The only constraints that can be assumed about processor B is that heart-beat messages sent by processor A will not be re-sent in a different order by processor B. However, there can be arbitrary delays in the re-sending of messages, and messages may be lost. Thus, every heart-beat message sent by processor A contains some information that uniquely identifies it, and the HBM software keeps track of the time that it was sent. When a message is received from processor B, the HBM software calculates the delay between the send-time and the receive-time of the message. Because of memory and performance limitations, the HBM software bounds the number of outstanding heart-beat messages (sent but not yet received) being tracked by keeping a small fixed-size array of messages sent to processor B. Each heart-beat message is uniquely identified by its array index and a timestamp corresponding to when it was sent; the receipt of the corresponding message is marked in the array

index. The HBM cycles through this fixed-size array when sending heart-beat messages. Upon visiting a particular array index, if the HBM software finds that the corresponding message has not yet been received, it is considered to be lost.

In order to avoid high sensitivity to delays/loss of individual messages, the HBM software is structured in three sequential, mutually exclusive, stages. The first stage indicates that resource suspension has not been triggered in the recent past, the second stage serves to dampen the sensitivity of the HBM to individual delays for a period of time, and the third stage is intentionally sensitive to all delays. After spending a fixed period of time in the third stage, the software re-enters the first stage. We note that resource suspension can be triggered only in the third stage. The algorithm used by the HBM software to determine whether resource suspension should be triggered is described in [10].

The decision to trigger resource suspension involves some tradeoffs. Calls routed over processor B may not behave reliably in the face of unacceptable propagation delays; on the other hand, resource suspension may cause many new calls to be blocked. Clearly, end-users may become irate in either situation; furthermore, switch operators are required by law to report every extended occurrence of call blocking to the Federal Communications Commission. Hence, the HBM software is carefully engineered to achieve a reasonable balance between switch reliability and switch capacity.

In the past, the HBM software in this switch had come under scrutiny because of discoveries in the field that resource suspension was occurring too frequently and resulting in a significant decrease in network capacity. The development team had obtained mid-stream timed trace observations of the system running in the field, and had attempted to reverse-engineer the software to better understand the underlying executions of these timed traces. As a research-development collaboration, we had in earlier work [10] used VeriSoft, a dynamic state-space exploration tool, to analyze the behavior of the HBM software of this switch against its intended properties, and our analysis had successfully revealed flaws in the software that were subsequently corrected by the development team. However, our analysis suffered from three drawbacks: (a) it abstracted away from real-time (b) it only performed analysis from the initial state of the system, and did not support mid-stream analysis, and (c) it required a significant investment in implementing a separate environment for each of the properties, in order to perform dynamic analysis of the system. These drawbacks form the basis of our motivation to use the HBM of this switch as a case study for timed automata-based passive testing.

4. PASSIVE TESTING OF THE HEART-BEAT MONITOR

In order to apply the timed automata-based testing framework described in Section 2 to the HBM software, we first specified as timed automata the intended real-time properties of the HBM, and then used the real-time model checker UPPAAL [7] to perform passive mid-stream monitoring of the HBM software against these properties.

4.1 Intended Properties of the HBM Software

In order to describe the intended properties of the HBM software, we use the following definitions and notation. We cannot reveal the actual values of the constants d_{ontime}, d_{stage2}, d_{period}, a_1, a_2, b below because of proprietary considerations, but note that $\lceil d_{stage2}/d_{period} \rceil < 10$, $a_2 < a_1 < 20$, and $b < 10$. First, we define d_{period} as the fixed period between successive heart-beat messages sent by the HBM. Second, we say that the *propagation delay* of a message that is sent but never received by the HBM is ∞. Otherwise, the propagation delay is $t_2 - t_1$, where t_1 is the time the message was sent by the HBM, and t_2 is the time it was received by the HBM. Third, we consider a message to be *on time* iff its propagation delay is less than or equal to d_{ontime}, where d_{ontime} is a non-zero integer constant strictly less than d_{period}. We consider a message to be *slightly late* iff its propagation delay is strictly between d_{ontime} and d_{period}. We consider a message to be *late/lost* iff its propagation delay is strictly greater than d_{ontime}.

The HBM software is intended to satisfy the following properties.

1. If no heart-beat messages return to processor A from the time that Stage 1 was last entered, then resource suspension is not triggered within $a_1 \times d_{period}$ time units since Stage 1 was last entered. Furthermore, resource suspension is triggered within $(a_1 + 2) \times d_{period}$ time units from the time that Stage 1 was last entered.

2. If B resends every message *slightly late* from the time that Stage 1 was last entered, then resource suspension is triggered within $(a_2 + 2) \times d_{period}$ time units from the time that Stage 1 was last entered.

3. Resource suspension is not triggered within $a_2 \times d_{period}$ time units since the last time Stage 1 was entered.

4. Resource suspension is not triggered before b messages have been late/lost since the last time Stage 1 was entered.

5. If messages strictly alternate between being *slightly late* and *on time* from the last time that Stage 1 was entered, resource suspension will never be triggered.

6. Stage 2 is never exited before d_{stage2} time units from the time it is entered. Furthermore, it is exited within $d_{stage2} + (3 \times d_{period})$ time units.

4.2 Experimental Framework and Results

In our approach, we used the real-time model checker UPPAAL as an off-line engine for passive testing. UPPAAL supports the description of networks (*i.e.*, parallel composition) of timed automata, and automatically performs verification of these timed automata specifications against properties specified in a real-time temporal logic.

In our application, we used UPPAAL to passively test that the interval-timed traces of the actual HBM code, observed mid-stream during the execution of the code, satisfy its desired properties. We note that the language of these properties are timed-prefix-closed, timed-suffix-closed sets of interval-timed traces, and hence the properties are passively testable. Thus, as described in Section 2, passive testing corresponds to checking of membership of the observed interval-timed traces of the HBM in the language of these properties.

Our application used the following steps:

1. We described the negation of each of the properties in Section 4.1 as timed automata in the UPPAAL specification language; in particular, the language of each timed automata is precisely the set of interval-timed traces (over the alphabet of stage changes in the HBM and the arrival of messages) that violates the corresponding property.

2. We instrumented the HBM code to generate an event when the HBM code transitions between its sequential stages.

3. We implemented in C a model of processor B that randomly either loses messages or sends messages back to the HBM on-time, slightly late, or with delays of random numbers of multiple intervals, while preserving the order of messages received from the HBM (as per the assumptions described in Section 3).

4. We ran the HBM code and the environment code concurrently as Solaris processes. The execution generated traces (*i.e.* sequences) of events corresponding to the sending and receiving of messages by the HBM, together with stage changes in the HBM (as per step 2 above), as well as the real-time delay between successive events. Consistent with our motivation for passive testing, we started observing each interval-timed trace from a random point mid-stream during its execution.[2] Each mid-stream interval-timed trace was bounded by a specific, pre-determined length, or by the HBM reaching a state where resource suspension was triggered.

5. For each generated interval-timed trace, we automatically checked that the trace satisfies all the intended properties of the HBM described in Section 4.1. In particular, we automatically constructed a UPPAAL timed automaton whose language is exactly the singleton set consisting of the given interval-timed trace. We then composed this timed automata in parallel with each of the specified timed automata from step 1 above. We then used the model checking capabilities of UPPAAL to check reachability of an accepting state of this composed timed automata. The UPPAAL semantics of parallel composition ensure that this is equivalent to intersection of the languages of these timed automata, and hence to checking whether the given interval-timed trace is contained in the language of the intended properties.

We generated 500 mid-stream interval-timed traces from this switching code, and checked them using the above approach. Consistent with our previous analysis using VeriSoft, we found that Property 6 is indeed violated by the code: that is, our passive testing framework reported that some of the observed mid-stream traces violated this property. Since the environment choses randomly whether to lose messages, or how much to delay them, most of the 500 traces likely did not satisfy the preconditions of the properties, and did not reveal any further property violations.

We thus constrained our environment so that it randomly chose among patterns of message delays/losses that satisfy the preconditions of the properties, such as all messages being lost, all messages being slightly delayed, all messages being on-time, all messages alternating between being on-time and being slightly late (for this last one, preceded by specific short patterns of earlier behavior in some cases). This constrained environment was constructed to increase the chances of revealing other property violations, and to strengthen our confidence in the satisfaction of properties for which violations were not found. Running 100 tests using this constrained environment, showed that both Property 5 and Property 6 were violated, while the remaining properties were not violated, consistent with our earlier analysis using VeriSoft.

[2]Since all of the properties are triggered by stage changes, we made the optimization that the mid-stream observations begin at a randomly chosen stage change.

5. CONCLUSIONS AND FUTURE WORK

We have presented a formal framework for passive mid-stream monitoring of real-time properties, together with an implementation of this framework using the UPPAAL model checker for timed automata. To demonstrate the feasibility of using this approach for actual systems, we have applied our approach to the analysis of some fault tolerant software in a telecommunications switch developed at Lucent Technologies. Since specifications in logics can often be easier to understand for testing practitioners, we plan to identify fragments of real-time logics [12] that define passively testable properties. We also plan to extend our implementation in several ways. First, we plan to implement on-line mid-stream monitoring, using timed automata observers [8] or on-line (timed) verification tools such as T-UPPAAL [16]. Second, we plan to explore the applicability of automated instrumentation of system implementations such as that described in [13] to facilitate run-time observation of timed behavior; in our case study, we had done some of this instrumentation manually.

6. REFERENCES

[1] R. Alur, C. Courcoubetis, and D. L. Dill. Model-checking for real-time systems. In *Logic in Computer Science*, 1990.

[2] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[3] R. Alur, L. Fix, and T. Henzinger. Event-Clock Automata: A Determinizable Class of Timed Automata. *Theoretical Computer Science*, 211:253–273, 1999.

[4] R. Alur, R. Kurshan, and M. Viswanathan. Membership problems for timed and hybrid automata. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, 1998.

[5] R. Alur and P. Madhusudan. Decision problems for timed automata: A survey. In *4th Intl. School on Formal Methods for Computer, Communication, and Software Systems: Real Time*, 2004.

[6] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In B. Steffen and G. Levi, editors, *VMCAI*, volume 2937 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2004.

[7] G. Behrmann, A. David, and K. G. Larsen. A tutorial on uppaal. In M. Bernardo and F. Corradini, editors, *SFM*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004.

[8] S. Bensalem, M. Bozga, M. Krichen, and S. Tripakis. Testing conformance of real-time applications with automatic generation of observers. In *Runtime Verification*, 2004.

[9] R. Cardell-Oliver. Conformance tests for real-time systems with timed automata specifications. *Formal Asp. Comput.*, 12(5):350–371, 2000.

[10] P. Godefroid, R. S. Hanmer, and L. J. Jagadeesan. Model checking without a model: An analysis of the heart-beat monitor of a telephone switch using verisoft. In *ISSTA*, pages 124–133, 1998.

[11] T. Henzinger, Z. Manna, and A. Pnueli. Temporal proof methodologies for timed transition systems. *Information and Computation*, 112:273–337, 1994.

[12] T. A. Henzinger. It's about time: Real-time logics reviewed. In D. Sangiorgi and R. de Simone, editors, *CONCUR*, volume 1466 of *Lecture Notes in Computer Science*, pages 439–454. Springer, 1998.

[13] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky. Formally specified monitoring of temporal properties. In *European Conference on Real-Time Systems*, 1999.

[14] M. Krichen and S. Tripakis. Black-box conformance testing for real-time systems. In S. Graf and L. Mounier, editors, *SPIN*, volume 2989 of *Lecture Notes in Computer Science*, pages 109–126. Springer, 2004.

[15] D. Lee, A. Netravali, K. Sabnani, B. Sugla, and A. John. Passive testing and applications to network management. In *Proceedings of the IEEE International Conference on Network Protocols*, pages 113–122, October 1997.

[16] M. Mikucionis, K. G. Larsen, and B. Nielsen. T-uppaal: Online model-based testing of real-time systems. In *ASE*, pages 396–397. IEEE Computer Society, 2004.

[17] A. N. Netravali, K. K. Sabnani, and R. Viswanathan. Correct Passive Testing Algorithms and Complete Fault Coverage. In *23rd IFIP International Conference on Formal Techniques for Networked and Distributed Systems (FORTE)*, pages 303–318, 2003.

[18] V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, December 1999.

[19] D. K. Peters and D. L. Parnas. Requirements-based monitors for real-time systems. *IEEE Trans. Software Eng.*, 28(2):146–158, 2002.

[20] M. Ranum, K. Landfield, M. Stolarchuk, M. Sienkiewicz, A. Lambeth, and E. Wall. Implementing a generalized tool for network monitoring. In *Proceedings of the Eleventh Systems Administration Conference (LISA XI)*, pages 1–8, 1997.

APPENDIX

A. SOME PROOFS

PROOF. (of Lemma 2.14).
The ($\Rightarrow$) direction trivially holds. For the ($\Leftarrow$) direction, consider any σ', σ with $\sigma' \oplus \sigma \in [\![I]\!]$. Then $\sigma' \oplus \sigma \in [\![I']\!]$ for some $I' \models \mathcal{P}$. Hence $\sigma \in PT(\mathcal{P})$. By the closure property of being passively testable, we have that $I \models \mathcal{P}$. □

PROOF. (of Theorem 2.16).

($\Rightarrow$) Condition (1) follows from Lemma 2.14. To establish Condition (2), consider any σ', σ with $\sigma' \oplus \sigma \in L_{\mathcal{P}}$; we need to show that $\sigma \in L_{\mathcal{P}}$. By (Axiom II) of Definition 2.4, we have an implementation I_σ with the characteristic property that $[\![I_\sigma]\!]$ is the set of all timed prefixes of σ. We will show that $I_\sigma \models \mathcal{P}$ and hence $\sigma \in L_{\mathcal{P}}$. Consider any σ_1, σ_2 such that $\sigma_1 \oplus \sigma_2 \in [\![I_\sigma]\!]$. By the characteristic property of I_σ this means that $\sigma_1 \oplus \sigma_2 \oplus \sigma_3 = \sigma$ for some trace σ_3. Since $\sigma' \oplus \sigma \in L_{\mathcal{P}}$, we have an $I \models \mathcal{P}$ with $\sigma' \oplus \sigma \in [\![I]\!]$. For $\sigma'' = \sigma' \oplus \sigma_1$, we have that $\sigma'' \oplus \sigma_2 \oplus \sigma_3 \in [\![I]\!]$. By (Axiom I) of Definition 2.4, we have that $\sigma'' \oplus \sigma_2 \in [\![I]\!]$ and $I \models \mathcal{P}$. From Definition 2.7, we therefore have that $\sigma_2 \in PT(\mathcal{P})$. By the closure condition of passive testability it then follows that $I_\sigma \models \mathcal{P}$.

($\Leftarrow$) We need to show that $\mathcal{P}$ admits complete coverage under passive testing assuming Conditions (1) and (2). Consider any I such that for all traces σ', σ with $\sigma' \oplus \sigma \in [\![I]\!]$ we have that $\sigma \in PT(\mathcal{P})$. We will show that $[\![I]\!] \subseteq L_{\mathcal{P}}$, and from Condition (1), it follows that $I \models \mathcal{P}$ thereby establishing the closure requirement of passive testability. Consider any $\sigma \in [\![I]\!]$. Since $0 \oplus \sigma \in [\![I]\!]$, we have that $\sigma \in PT(\mathcal{P})$. Therefore we have an $I' \models \mathcal{P}$ and σ such that $\sigma' \oplus \sigma \in [\![I']\!]$. Since $I' \models \mathcal{P}$, we have that $\sigma' \oplus \sigma \in L_{\mathcal{P}}$. From Condition (2), we get that $\sigma \in L_{\mathcal{P}}$, thus establishing that $[\![I]\!] \subseteq L_{\mathcal{P}}$.

□

PROOF. (of Theorem 2.18).
Consider any $\sigma \in PT(\mathcal{P})$. Then we have a σ' and an $I \models \mathcal{P}$ with $\sigma' \oplus \sigma \in [\![I]\!]$. Since $I \models \mathcal{P}$, we have that $\sigma' \oplus \sigma \in L_{\mathcal{P}}$. Since $\mathcal{P}$ is passively testable, by Theorem 2.16, $L_{\mathcal{P}}$ is timed suffix-closed and hence $\sigma \in L_{\mathcal{P}}$. This shows that $PT(\mathcal{P}) \subseteq L_{\mathcal{P}}$.

The other direction of inclusion is true for any property $\mathcal{P}$. Consider any $\sigma \in L_{\mathcal{P}}$. We therefore have an $I \models \mathcal{P}$ with $\sigma \in [\![I]\!]$. For $\sigma' = 0$, we have that $0 \oplus \sigma \in [\![I]\!]$ for $I \models \mathcal{P}$. Using Definition 2.7, it follows that $\sigma \in PT(\mathcal{P})$. □